# OpenJDK and OracleJDK: Which compiler generates faster, more memory-efficient compiled scientific and numerical computing Java programs?

**Quan Chau**

**8 December, 2017**

## Abstract

The project focuses on using scientific benchmarks to compare two prominent Java compilers included in two Java Development Kits (JDK), OpenJDK and OracleJDK, based on the efficiency of the compiled programs. Installing two JDKs on two Ubuntu systems (with similar configurations) using VirtualBox, we ran each benchmark several times on each system by the built-in Linux command line function to get results about runtime and memory usage.

Overall, the result shows that programs compiled by OpenJDK compiler had lower efficiency in memory usage in both RAM and hard drive than programs compiled by the OracleJDK compiler. Furthermore, the overall runtime result points out that OpenJDK compiled programs also take more time to run, although it may not be obvious if we judge the benchmarks separately.

## 1. Introduction

By benchmarking Java compiled programs, we will determine which compiler generates the resulted compiled programs that require the least amount of time and space in RAM and in the computer's hard drive to implement scientific and numerical computation.

This project allows one to use the best compiler when implementing complicated scientific Java programs, thus maximizing efficiency. It is important to select the right compiler before writing a Java program since the compiler affects the performance the most in Java Virtual Machine [8]. The efficiency of the compiler affects both the instruction count and average cycles per instructions of the resulting program when it translates high-level languages into computer instructions. Therefore, choosing which compiler to use that fits a specific numerical program is a crucial step for anyone who is working on Java applications.

In addition, our decision to test Java compilers is also important due to the fact Dickinson College has a Java-based Computer Science curriculum. Our benchmarking result will provide additional information to Dickinson students to allow them to decide which compiler to use when compiling methods with Java.

In this project, we consider two popular Java compilers in the industry. The former is of OpenJDK (Open Java Development Kit), which is a free and open source implementation of the Java Platform, Standard Edition (Java SE) [14]. The latter is of OracleJDK (Oracle Java Development Kit), which builds on OpenJDK by adding more features such as deployment code including Oracle's implementation of Java Plugin and Java WebStart [9].

### *1.1. Performance aspects*

There are three aspects of performance to be measured in this project, namely runtime, memory usage and compiled file size.

1. Runtime (part of the CPU time measured in seconds): We use CPU time to determine the amount of time a compiled program takes within a CPU when dealing with a scientific and

numerical computation. Runtime is important due to the fact that developers need to test the programs multiple times and that users often favor the fastest programs.

2. Memory usage (Resident Set Size measured in kilobytes): We use Resident Set Size because it reflects the amount of space of physical memory (RAM) held by a specific process, which is ideal to measure the memory used when the benchmarks are run [12]. When memory requirement is too big after compilation, the compiled program may exhibit degradation. In such case, failures occur due to memory run-out and can result in the inefficient use of memory that can affect a larger system.

3. Compiled file size (memory held in hard drive, measured in bytes): In fact, the compile file size also reflects a part of the amount of memory a compiled file requires. The difference is that this aspect focuses on the size of a static file, which means how efficiently a compiler can generate a file that takes the least amount of space on the computer's hard drive.

## 2. Methods

### *2.1. Benchmarks*

In this project, most of our benchmarks are chosen from SciMark 2.0, which is a set of Java benchmarks for scientific and numerical computing. SciMark 2.0 consists of five computational kernels: FFT, Gauss-Seidel relaxation, Sparse matrix-multiply, Monte Carlo integration and dense LU factorization. These kernels are chosen to provide an indication of how well the underlying JVM/JITs perform on applications utilizing these types of algorithms. The problems sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM/JIT and CPU issues [11]. In addition, we also choose Binary Trees benchmark for

this project to provide objective results instead of merely focusing on SciMark. The Binary Trees benchmark comes from a project called “The Computer Language Benchmarks Game”, which has been previously used to compare programming languages in terms of time and memory usage. Since it is also a scientific benchmark, we want to see if there are significant differences between the results generated by this benchmark and those generated by the SciMark benchmarks.

The brief descriptions of 6 benchmarks we use for this project are as follows:

1. Binary Trees Benchmark: Allocates and deallocates many binary trees, then prints the time required to allocate and collect balanced binary trees of various sizes. Smaller trees result in shorter object lifetimes [7].

2. Fast Fourier Transform: Performs a one-dimensional forward transform of 4K complex numbers. This benchmark exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions. The first section performs the bit-reversal portion and the second performs the actual Nlog(N) computational steps [11].

3. Montel Carlo Integration: Approximates the value of $\pi$ by computing the integral of the quarter circle $y = \text{sqrt}(1 - x^2)$ on [0,1]. It chooses random points with the unit square and compute the ratio of those within the circle. The algorithm exercises random-number generators, synchronized function calls, and function inlining [11].

4. Jacobi Successive Over-relaxation (SOR): Implemented on a 100x100 grid exercises typical access paterns in finite difference applications, for example, solving Laplace’s equation in 2D with Drichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns where each A(i,j) is assigned an average weighting of its four nearest neighbors [11].

5. Dense LU Matrix Factorization: Computes the LU factorization of a dense 100x100 matrix using partial pivoting. Exercises linear algebra kernels (BLAS) and dense matrix operations. The algorithm is the right-looking version of LU with rank-1 updates [11].

6. Sparse Matrix Multiplication: Uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references. A 1,000 x 1,000 sparse matrix with 5,000 nonzeroes is used [11].

These benchmarks are used mainly because they include a variety of scientific and numerical computation, which is the focus of this project. Moreover, they are solid Java programs that can be run with simple setup and have been tested by other programmers to generate sets of results about runtime and memory usage. Therefore, based on the existing results and explanation in the sources we found, the results from these benchmarks are reliable and can be used to serve the purpose of this project.

### *2.2. System version and Configuration*

To set up two equal running environments for two systems, we used the machine and softwares with configuration information in Table 2.1 to support our project.

Table 2.1: System version and Configuration information

| | **Description** |
|---|---|
| **Computer** | iMac 2.7 GHz Intel Core i5 with 8 GB 1600 MHz DDR3 |
| **Virtual Box** | Version 5.1.0 r108711 |
| **Operating Systems** | x64 Ubuntu 4GB base memory and<br>100.0 GB fixed size storage |

| | |
|---|---|
| **IDE** | Eclipse Oxygen for win64 |
| **Compilers** | openjdk-8-jre and openjdk-8-jdk<br>Oracle jdk1.8.0_151 |

### *2.3. Installation*

We created two versions of Ubuntu with the above configuration in VirtualBox. On the first system, called System 1, which represents the OpenJDK Compiler, we follow the instruction in Table 2.2.

Table 2.2: Steps to install OpenJDK on System 1

| **Installing OpenJDK** | |
|---|---|
| **1** | Download and install Eclipse Oxygen from<br>*https://www.eclipse.org/downloads/download.php?file=/oomph/epp/oxygen/R/eclipse-inst-win64.exe* |
| **2** | Download and install openjdk-8-jre and openjdk-8-jdk by running two command lines:[1]<br>*$ sudo apt-get install openjdk-8-jre*<br>*$ sudo apt-get install openjdk-8-jdk* |

On the second system, called System 2, which represents the OracleJDK Compiler, we walk through the steps in Table 2.3.

Table 2.3: Steps to install OracleJDK on System 2

| **Installing OracleJDK** | |
|---|---|
| **1** | Download and install Eclipse Oxygen from *https://www.eclipse.org/downloads/download.php?file=/oomph/epp/oxygen/R/eclipse-inst-win64.exe* |
| **2** | Download OracleJDK jdk1.8.0_151 from *http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html* |
| **3** | Install OracleJDK jdk1.8.0_151 by running these command lines: [2]<br><br>*$ tar zxvf jdk-8u151-linux-x64.tar.gz*<br>*$ sudo mkdir /usr/lib/jvm*<br>*$ sudo mv jdk1.8.0_151 /usr/lib/jvm*<br><br>*$ sudo update-alternatives –install /usr/bin/java java /usr/lib/jvm/jdk1.8.0_151/bin/java 1*<br>*$ sudo update-alternatives –install /usr/bin/javac javac /usr/lib/jvm/jdk1.8.0_151/bin/javac 1*<br>*$ sudo update-alternatives –install /usr/bin/javaws javaws /usr/lib/jvm/jdk1.8.0_151/bin/javaws 1* |

### *2.4. Generating results*

We downloaded the benchmarks' source code and imported it into Eclipse. Then the programs were compiled through Eclipse by the installed compiler. To run the compiled programs, we opened the terminal window and directed to the folder containing the *.class* files, which is normally the *bin* folder within the Eclipse project. In the terminal window, we ran the following command:

*usr/bin/time –v java [.class fileName]*

What this command does is generating a list of information about runtime, memory usage and other attributes of the programs. [10] Here is an example of what appears when running this command:

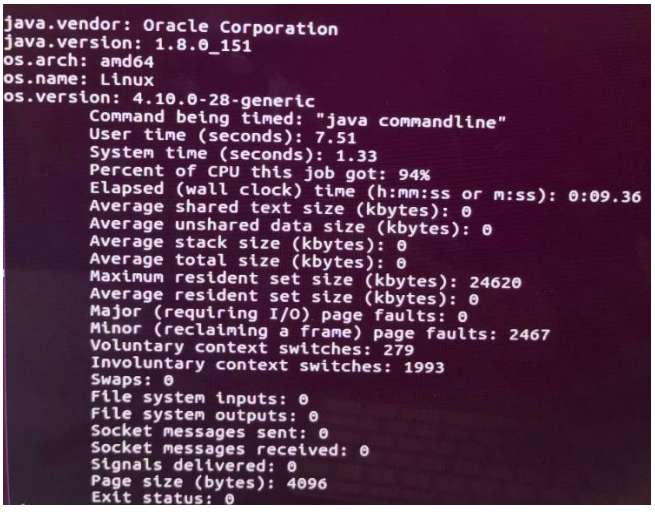


Figure 2.1: An example of results generated by *usr/bin/time –v java FileName* command

In order to get the information associated with the performance aspects mentioned above, we analyze these data:

1. Runtime (measured in seconds): Since our project focuses mainly on CPU time, we get the results for this aspect by noting down the *User time*, which reflects a part of the CPU time [5].

2. Memory usage (measured in kilobytes): To get the memory usage, we relied on the information from *Resident set size* section [15].

3. Compiled file size (measured in bytes): The compiled file size is shown in the properties of the *.class* file in the *bin* folder of the Eclipse project.

In order to ensure the data's consistency, for runtime and memory usage, we ran each benchmark five times and computed the arithmetic mean for each system. Since the compiled file size is relatively consistent, we only compiled the file once through Eclipse and get the size of the resulted *.class* files. Table 2.4 provides an example of the data we collected from running the benchmarking programs.

Table 2.4: Data collected from running Fast Fourier Transform on System 1

| | **Runtime (s)** | **Memory usage (KB)** | **Compiled file size (bytes)** |
|---|---|---|---|
| **1st trial** | 7.18 | 25,660 | 3,977 |
| **2nd trial** | 7.18 | 27,592 | |
| **3rd trial** | 7.18 | 27,496 | |
| **4th trial** | 7.42 | 27,696 | |
| **5th trial** | 7.42 | 27,412 | |

With the data collected, we then calculated the composite normalized results by averaging all the data for each benchmark using arithmetic mean since the data is not in forms of rates or percentages. After that, we normalized them to System 1 using equation 2.1. The normalized results were then composited using geometric mean by equation 2.2.

$$System\ A = \frac{System\ 1}{System\ A} \tag{2.1}$$

$$Geometric\ mean\ of\ n\ values = \sqrt[n]{x_1.x_2 \dots x_n}. \tag{2.2}$$

## 3. Results

From now on, we will refer to the system with OpenJDK as System 1 and the system with OracleJDK as System 2. Our result description is divided into two parts: Average data and Normalized composite result.

### *3.1. Average data*

Table 3.1 represents the average results of the data collected from each benchmark for three performance aspects.

Table 3.1: Arithmetic mean of data after running each benchmark 5 times

| | **Runtime (s)** | | **Memory usage (KB)** | | **Compiled File Size (Bytes)** | |
|---|---|---|---|---|---|---|
| | System 1 | System 2 | System 1 | System 2 | System 1 | System 2 |
| **Binary Tree** | 0.132 | 0.146 | 31,161.6 | 29,424 | 3,641 | 3,449 |
| **Fast Fourier Transform** | 7.276 | 7.196 | 27,171.2 | 24,924 | 3,977 | 3,951 |
| **Monte Carlo** | 7.182 | 7.128 | 27,336 | 24,840.8 | 817 | 765 |
| **Dense LU Matrix Factorization** | 5.462 | 5.466 | 28,322.4 | 25,671.2 | 3,453 | 3,422 |
| **SOR** | 13.022 | 7.602 | 29,749.6 | 24,675.2 | 1,145 | 1,119 |
| **Sparse Matrix Multiplication** | 7.726 | 7.646 | 26,700 | 24,705.6 | 1,012 | 986 |

Since runtime, memory usage and compiled file size were generated in seconds, kilobytes and bytes respectively, and it is ideal to have a fast, less space-consuming and small-sized compiled programs, smaller data accounts for better performance.

### *3.2. Normalized composite result*

The above data were then normalized to System 1 and the geometric mean was calculated for further comparison, which gave the below normalized composite results. The reason why we used geometric mean to average normalized results is to ensure consistency and not be strongly affected by one exceptional case due to its multiplicative property, which can be stated simply that the mean of the products equals the product of the mean [6].

In addition, we should note that the results are normalized to System 1 so that the larger the numbers are in System 2, the better the compiled programs in System 2 perform compared to those in System 1.

#### *3.2.1 Runtime*

Table 3.2: Normalized composite result on runtime of two systems

| **Runtime** | | |
|---|---|---|
| | System 1 | System 2 |
| **Binary Tree** | 1.000 | 0.904 |
| **Fast Fourier Transform** | 1.000 | 1.011 |
| **Monte Carlo** | 1.000 | 1.008 |
| **Dense LU Matrix Factorization** | 1.000 | 0.999 |
| **SOR** | 1.000 | 1.713 |
| **Sparse Matrix Multiplication** | 1.000 | 1.010 |
| **Geometric Mean** | 1.000 | 1.081 |

In Figure 3.1, although when running the first and fourth benchmarks, System 1's compiled programs show higher efficiency compared to those of System 2, the programs are either slightly or dramatically lower than those of System 2 when we take the other benchmarks into consideration. However, since the cases in which the program in System

1 is better are trivial, the geometric mean of the normalized composite results still show higher efficiency for System 2.

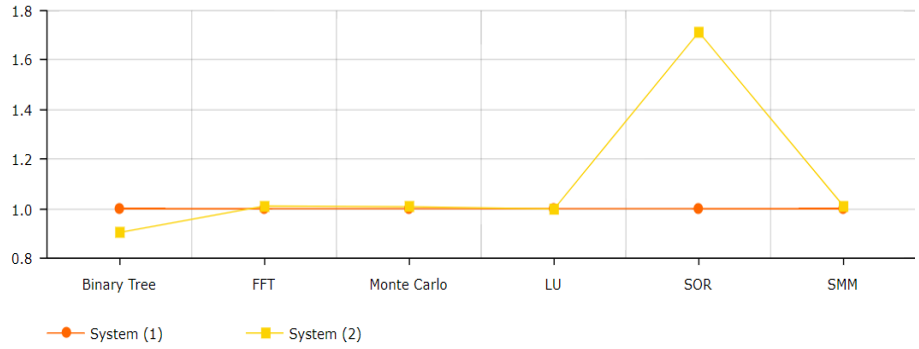


Figure 3.1: Normalized composite result on runtime of two systems

### *3.2.2 Memory Usage*

Table 3.3: Normalized composite result on memory usage of two systems

| **Memory Usage** | | |
|---|---|---|
| | System 1 | System 2 |
| **Binary Tree** | 1.000 | 1.059 |
| **Fast Fourier Transform** | 1.000 | 1.090 |
| **Monte Carlo** | 1.000 | 1.1 |
| **Dense LU Matrix Factorization** | 1.000 | 1.103 |
| **SOR** | 1.000 | 1.205 |
| **Sparse Matrix Multiplication** | 1.000 | 1.08 |
| **Geometric Mean** | 1.000 | 1.105 |

It can be seen from Figure 3.2 that using the same source code, the compiled programs of System 1 take more space in RAM while running.

It is also worth noting from the SOR benchmark that the result generated from System 1 shows much less efficiency than that from System 2, which is similar to the result we got for runtime. Another interesting point is that although the gaps between

System 1's results and System 2's results vary, the general trends of System 2 in Figure 3.1 and Figure 3.2 are seemingly identical. From that observation, we can conclude that there is a correlation between time efficiency and memory efficiency.

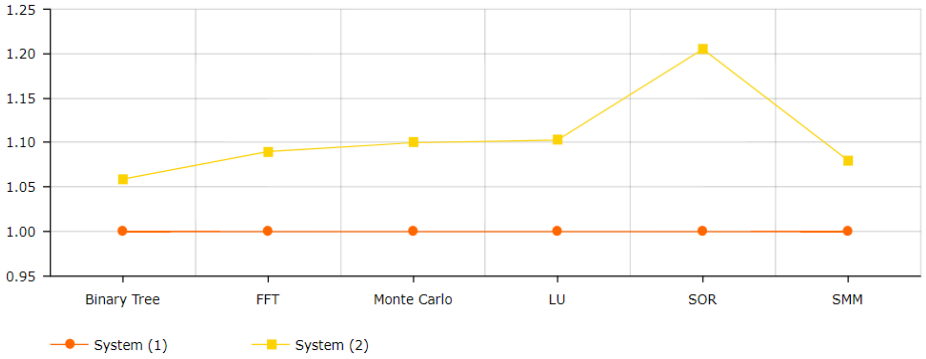


Figure 3.2: Normalized composite result on memory usage of two systems

### *3.2.3 Compiled file size*

Table 3.4: Normalized composite result on compiled file size of two systems

| **Compiled File Size** | | |
|---|---|---|
| | System 1 | System 2 |
| **Binary Tree** | 1.000 | 1.04 |
| **Fast Fourier Transform** | 1.000 | 1.006 |
| **Monte Carlo** | 1.000 | 1.067 |
| **Dense LU Matrix Factorization** | 1.000 | 1.009 |
| **SOR** | 1.000 | 1.023 |
| **Sparse Matrix Multiplication** | 1.000 | 1.026 |
| **Geometric Mean** | 1.000 | 1.028 |

Similar to other performance aspects, the compiled programs that take less runtime and space in RAM also take less space in hard drive to store the files. To be more specific, as shown in figure 3.3, every benchmark we have after being compiled in System 1 takes more space to store the *.class* files than after being compiled in System 2.

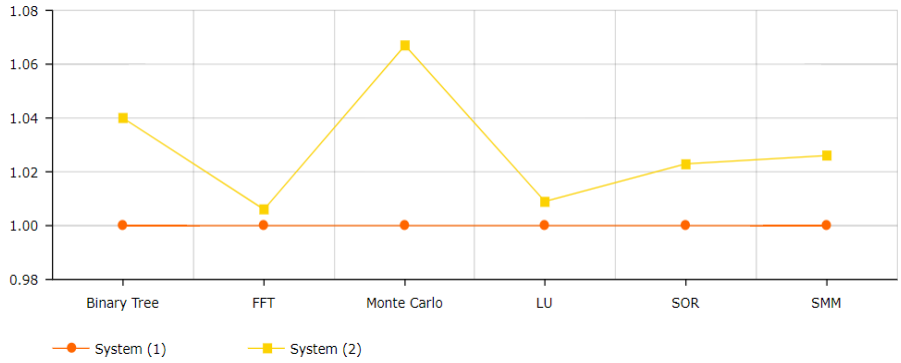


Figure 3.3: Normalized composite result on compiled file size of two systems

## 4. Discussion

### *4.1 Conclusion*

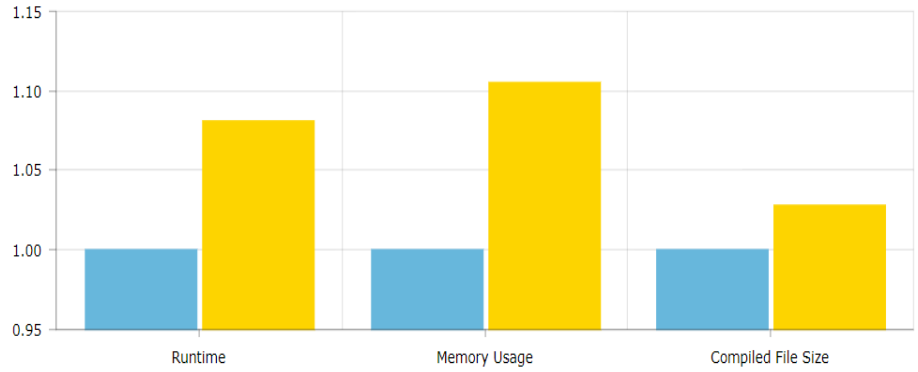


Figure 3.4: Geometric mean of three performance aspects between two systems

Overall, from previous results and the geometric mean of the normalized composite results (Figure 3.4), we can conclude on three points:

1. System 2 performs better in every performance aspect that we consider in this project. Moreover, the results also reflect the similarity between the figure of runtime and that of

memory usage. To be more specific, when we take any benchmark into consideration, if the compiled program in one system takes longer to run in the CPU than in another system, it is likely that the program will also use more memory in the computer's RAM. More interestingly, it is also worth noting that a benchmark with larger or smaller normalized composite results than other benchmarks in runtime also has larger or smaller normalized composite results in memory usage as well.

2. While programs compiled by OpenJDK shows 8.1% and 10.5% better performance than programs compiled by OracleJDK in terms of runtime and memory usage respectively, the latter's performance regarding compiled file size is just slightly better (2.8%) than that of the former. From this observation, if all three aspects are taken into consideration when choosing compilers, compiled file size is less important compared to the other two aspects.

3. Comparing Binary Tree benchmark's data to those of other benchmarks, we can see that there are not much difference between them. In particular, although for runtime, Binary Tree benchmark shows a slight difference in which System 1 with OpenJDK compiler runs faster than System 2 with OracleJDK compiler, the results are similar regarding memory usage and compiled file size.

In brief, according to the obtained results and depending on the characteristics of the scientific and numerical computing programs, we can decide to use OracleJDK if the only aspects we focus on are just runtime, memory usage, and compiled file size.

### *4.2 Limitations and further work*

The decision whether or not to choose a compiler is not limited to these three performance aspects, since some optimizing compilers are designed with optimizations such as loop

optimizations and controlling optimizations that minimize or maximize some attributes of an executable computer program [4, 15]. Future work includes generating results for a wider range of performance aspects and benchmarks so that we will be able to ultimately decide on which compilers we should choose according to our priority.

Since this project focuses mostly on benchmarks within a same source (SciMark), the reliability of the results is not guaranteed. Although in fact, these benchmarks compute different sets of data and have different behaviors, they are written by only one group of people. Instead, the result can be more objective and the decision can be more precisely made if we can take more benchmarks from different sources into account and see if the data they generate have the same pattern as the data we already have.

Moreover, there are also other limitations during our process of generating results. Firstly, since the CPU includes both System time and User time, in that User time is the amount of CPU time spent outside the kernel and System time is the amount of CPU time spent within the kernel [5], there may be a different set of results if we consider the CPU time as the sum of these two values. In addition, when the full amount of space required by a process exceeds the *Resident Set Size*, the remaining portion is typically stored in *swap.* Therefore, the results can be more precise if we also consider *swap space* as part of the memory usage, especially when dealing with large programs [12].

Another limitation of the results is that due to the nature of these benchmarks, the Java Virtual Machine (JVM) was not "properly" warmed up, which means the classes were not cached beforehand. In fact, when classes are pushed into the JVM cache, they will be accessible faster during runtime than when the requests are first made [3]. Although it can be seen that in the Binary Trees Benchmark, the process of warming up the JVM is implied through the creation of many

binary trees before giving the final results, additional work should be done to ensure that all the benchmarks actually warm up the JVM by loading a number of classes before starting the main program.

**Acknowledgements**

I would like to express my special thanks to Professor Braught for giving helpful advice on running the benchmakrs and helping my team walk through each phase of this project.

Secondly I would also like to thank my teammates Seongho Lee, Changsu Nam and Sam Hrncir for having greatly contributed to the final outcomes.

**References**

1. 쌍쌍바나나. “Linux(Ubuntu)에 Java설치 및 환경 설정하는 방법”, *Ourcstory*, 2016, <http://ourcstory.tistory.com/129>
2. 신씅. “우분투(Ubuntu) - Oracle JDK 설치하기”, *Sseungshin*, 2016, <http://sseungshin.tistory.com/68>
3. Baeldung. “How to warm up the JVM.”, *baeldung*, 2017, http://www.baeldung.com/java-jvm-warmup
4. Brais H. “Compilers – What every programmer should know about compiler optimizations”, *Microsoft Magazine*, Feb 2015, https://msdn.microsoft.com/en-us/magazine/dn904673.aspx

5. ConcrenedOfTunbridgeWells. “What do ‘real’, ‘user’ and ‘sys’ mean in the output of time1?”, *Stackoverflow,* 2017, <https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1>

6. Fleming, Philp J. & John J. Wallace. “How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results.”, *Communications of the ACM,* Edited by Edgar H. Sibley, vol. 29, no. 3, Mar. 1986.

7. Jarkko M. “binary-trees Java #7 program”, *The Computer Language Benchmarks Game,* <http://benchmarksgame.alioth.debian.org/u64q/binarytrees-description.html#binarytrees>

8. Oaks S. “Chapter 4: Working with the JIT Compiler”, *Safari Books Online,* <https://www.safaribooksonline.com/library/view/java-performance-the/9781449363512/ch04.html>

9. Radai. “Differences between Oracle JDK and Open JDK and garbage collection”*Stackoverflow,* 2014, < https://stackoverflow.com/questions/22358071/differences-between-oracle-jdk-and-open-jdk-and-garbage-collection>

10. Roche A. “/usr/bin/time: not the command you think you know”, *Hackernoon,* 2017, <https://hackernoon.com/usr-bin-time-not-the-command-you-think-you-know-34ac03e55cc3>

11. Roldan P. & Bruce M. “How fast is your Java platform for number crunching?”, *SciMark 2.0,* 2014, <http://math.nist.gov/scimark2/about.html>

12. Wikichip contributors. “Resident Set Size (RSS).” *Wikichip, Computer Engineering,* 11 Dec. 2015, < https://en.wikichip.org/wiki/resident_set_size>

13. Wikipedia contributors. "Resident set size." *Wikipedia, The Free Encyclopedia*, 24 Feb. 2017. Web. 21 Nov. 2017, <https://en.wikipedia.org/wiki/Resident_set_size>

14. Wikipedia contributors. "OpenJDK." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 15 Nov. 2017. Web. 5 Dec. 2017.

15. Wikipedia contributors. "Optimizing compiler." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 21 Nov. 2017. Web. 21 Nov. 2017, < https://en.wikipedia.org/wiki/Optimizing_compiler#Types_of_optimization>

**Appendix**

These are the revisions I made to my draft after the writing workshop:

1. In the introduction, I clarify more about OpenJDK and OracleJDK for readers who do not know that they are. I also added some ideas regarding the significance of this the decision to be made and how considering which Java compilers to use would be crucial.
2. I added more information to describe what my benchmarks are and how they work responding to the questions on the course website. Also, I listed out some reasons why I chose them and how they would contribute to the results of my project.
3. In the Methods section, I add a brief explanation about what I will do with the data collected, including how I normalize and composite the results. An example of the data collected in 5 trials is also provided for clarification.
4. I move the “Conclusion” part from Results to Discussion since my discussion should also include the decision I made and the link between the data collected and my decision.
5. I also added my point of view about using benchmarks from SciMark and using those from other sources. As a result, I found out some interesting similarities between the data and the limitation of my project.